

Precondition Inference for Peephole Optimizations in LLVM

David Menendez

Rutgers University
davemm@cs.rutgers.edu

Santosh Nagarakatte

Rutgers University
santosh.nagarakatte@cs.rutgers.edu

Abstract

Peephole optimizations are a common source of compiler bugs. Compiler developers typically transform an incorrect peephole optimization into a valid one by strengthening the precondition. This process is challenging and tedious. This paper proposes PInfer, a data-driven approach for inferring preconditions for peephole optimizations expressed in Alive. PInfer generates positive and negative examples for an optimization, enumerates predicates on-demand, and learns a set of predicates that separate the positive and negative examples. PInfer repeats this process until it finds a precondition that ensures the validity of the optimization. PInfer reports both the weakest precondition and a set of succinct partial preconditions to the developer. The PInfer prototype successfully generates either the partial precondition or the weakest precondition for 164 out of 174 peephole optimizations in the Alive suite. It also generates preconditions that are weaker than LLVM's precondition for 73 optimizations. We also demonstrate the applicability of this technique to generalize 54 concrete expression directed acyclic graphs generated by an LLVM IR-based super optimizer.

1. Introduction

LLVM is a widely used compiler both in industry and academia. The LLVM compiler performs a large number of semantics-preserving optimizations to attain the best possible performance. Among them, peephole optimizations are an integral part of LLVM. Peephole optimizations perform local rewriting of the code with a primary focus on algebraic simplifications. In LLVM, they match the input program with respect to a pattern and replace them with a equivalent set of instructions. They also clean up and canonicalize code, which can enable other optimizations. Peephole optimizations in LLVM are performed by the InstCombine and

the InstSimplify passes. They are a persistent source of LLVM bugs [24, 29, 49].

To address this problem, we have proposed Alive [29], a domain specific language to specify and verify peephole optimizations for LLVM. The Alive language is similar to the LLVM intermediate representation (IR). An Alive optimization has a source pattern and a target pattern, with an optional precondition (see Section 2 for details on the Alive language). The Alive interpreter checks the correctness of an optimization using satisfiability modulo theories (SMT) solvers. Alive has discovered numerous bugs and is currently used by LLVM developers [18, 27, 29, 40].

Alive prevents the inclusion of wrong optimizations in the LLVM compiler. It also provides counterexamples for wrong optimizations. However, developing a correct optimization when presented with a Alive counterexample can be tedious. To ensure correctness, a developer must exclude all inputs that make the optimization invalid. Developers typically accomplish this by strengthening the precondition of the optimization. To illustrate, let us consider the following peephole optimization (presented in Alive syntax), which was submitted as a C++-code patch to the LLVM compiler on July 27, 2014 [18].

```
Pre: isPowerOf2(C1 ^ C2)
%x = add %A, C1
%i = icmp ult %x, C3
%y = add %A, C2
%j = icmp ult %y, C3
%r = or %i, %j
=>
%and = and %A, ~(C1 ^ C2)
%lhs = add %and, umax(C1, C2)
%r = icmp ult %lhs, C3
```

The patch was rejected because Alive found it to be invalid and provided a counterexample. Subsequently, the developer submitted multiple incorrect patches to rectify it. Finally, the developer submitted a valid optimization on August 18, 2014, with the following precondition.

```
C1 u> C3 && C2 u> C3 && isPowerOf2(C1 ^ C2) &&
isPowerOf2(-C1 ^ -C2) &&
(-C1 ^ -C2) == ((C3-C1) ^ (C3-C2)) &&
abs(C1-C2) u> C3
```

Although the above precondition makes the optimization valid, it is also too strong as it rejects many possible valid valuations for the symbolic constants. A compiler writer usually makes subtle trade-offs between the valuations allowed by the precondition and the succinctness of the precondition. If the precondition is too strong (while being valid), it prevents the application of the optimization on a large number of programs. In contrast, a precondition with too many predicates can increase compilation time. A precondition is checked with respect to the input pattern during compilation and the optimization is applied only when the precondition is satisfied by the pattern. Further, compilation time is a concern because peephole optimizations are applied numerous times for the same input program. Hence, developers prefer succinct preconditions that have as few predicates as possible. Identifying appropriate preconditions for these optimizations is challenging and developers will benefit from techniques that assist them in this process.

This paper proposes *PIInfer*, a data-driven approach to identify appropriate preconditions for LLVM peephole optimizations expressed in Alive. We are inspired by prior data-driven approaches that attempt to infer preconditions for general purpose programs [13–15, 37, 41, 45]. The precondition language of LLVM peephole optimizations is pretty expressive (see Figure 1). The precondition primarily involves compile-time symbolic constants, constant functions, and predicates. Further, the developer typically does not know the set of predicates that constitute the precondition. Hence, our approach builds on *PIE* [37], which learns predicates on-demand, and addresses new challenges in the context of Alive/LLVM optimizations.

Our approach has three main tasks in learning a precondition for an Alive optimization. First, we have to design positive and negative data (examples) for an optimization. An example should contain valuations for symbolic constants. We should consider types, because Alive optimizations are parametric over types. An example is positive if the optimization is valid for all runtime variables when we substitute symbolic constants with concrete valuations. We also need to address compile-time undefined behavior that can arise with examples. We generate examples using random sampling of the input space of symbolic constants and with the help of SMT solvers (see Section 3.1).

Second, we have to learn a set of predicates to separate positive and negative examples. Like *PIE* [37], we enumerate predicates on-demand and test them on a small sample of the set of examples that are not completely separated with the current set of predicates. We accept the predicate for consideration when it separates the sample because it can be helpful in narrowing the search for future predicates (see Section 3.2).

Third, we have a Boolean formula learning stage to generate the precondition from a set of predicates. As compiler developers prefer succinct preconditions, we generate a par-

tial precondition even when we have not completely separated the positive and negative examples (see Section 3.3). The partial precondition accepts a subset of the positive examples while rejecting all negative examples. We generate a complete precondition when we have learned a set of predicates that completely separate the positive and negative examples. We check the validity of the learned precondition. We repeat the process until the precondition is valid and weakest possible for the optimization. We report both partial preconditions and the weakest precondition to the compiler developer.

PIInfer is built on the publicly-available Alive-NJ tool. We evaluated the *PIInfer* prototype for generating preconditions with the Alive suite of optimizations. Out of the 417 optimizations in the Alive suite, there are 174 optimizations that have a precondition. We generated the weakest precondition for 133 of them within 1000 seconds. We generate either partial or the weakest precondition for 164 out of the 174 optimizations. More importantly, *PIInfer* generates a weaker precondition than the existing LLVM precondition for 73 optimizations.

To further demonstrate the applicability of our approach, we used *PIInfer* to generalize concrete expression directed acyclic graphs generated by an LLVM IR-based super optimizer (*Souper*) [21]. The super-optimizer collects equivalent source and target directed acyclic graphs by examining numerous code bases. These DAGs have concrete values for the leaf nodes. We generalized them by replacing concrete constants with symbolic constants. We generalized a total of 71 optimizations that are expressible in Alive. We are able to generate preconditions for 54 of them.

Contributions. This paper:

- Proposes a methodology for inferring preconditions for LLVM peephole optimizations. It also addresses the challenges in generating examples, learning predicates on-demand, and learning a Boolean formula for generating the precondition.
- Proposes a methodology to generate not only the weakest but also partial yet succinct preconditions. All preconditions reported to the compiler developer are valid.
- Demonstrates that the *PIInfer* prototype generates either the partial or the weakest precondition for 164 optimizations. 73 of these preconditions are weaker than LLVM’s precondition.
- Shows that the proposed approach is also useful in generalizing the optimization patterns generated by an LLVM IR-based *Souper* optimizer.

2. Background on Alive

Alive [29] is a domain-specific language to specify and verify peephole optimizations in LLVM. The Alive interpreter checks the correctness of an Alive optimization by encoding it as constraints in theories amenable for automated reason-

ing with SMT solvers. The interpreter also generates C++ code when the optimization is correct. To enable adoption by LLVM developers, the Alive language is similar to the LLVM intermediate representation. Alive has found numerous bugs in the LLVM compiler [18, 29]. It has prevented many bugs in patches committed to LLVM [18, 27, 40]. Alive is open source. LLVM developers are actively using Alive to check the correctness of new optimizations (patches) submitted to LLVM. Although C++ code generation is not actively used, there are plans on replacing InstCombine with Alive-generated C++ code. Other projects have also built extensions to Alive to reason about the correctness of floating point optimizations (e.g., Alive-FP [34] and LifeJacket [36]). Given its adoption by developers, we describe our approach for precondition inference using Alive. We describe the Alive language and its verification process below.

Alive language. An Alive optimization is of the form $\text{source} \Rightarrow \text{target}$ with an optional precondition. Alive also supports memory operations in its experimental version. In the absence of memory operations, both the source and the target represent a collection of directed acyclic graphs (DAGs). The intermediate nodes in the directed acyclic graphs represent Alive (LLVM IR) instructions. Hence, the semantics of the LLVM IR instruction determines the semantics of the intermediate node. The leaves of the DAG are called input variables. Input variables represent arbitrary LLVM values such as results from other instructions, inline constants, and function parameters. In general, some of these values will not be known at compile time. Semantically, an Alive optimization replaces the DAG in the source by the DAG in the target. Hence, the root node of the respective DAGs should have the same name.

In contrast to the LLVM IR, Alive optimizations are parametric over integer types and have symbolic constants. Symbolic constants are compile-time input variables, and optimizations can compute with them at compile time. A sample Alive optimization is shown in Figure 4(a) and its DAG representation is shown in Figure 4(b). There are three input variables — X , C_1 , and C_2 , where X is a runtime input variable and C_1 and C_2 are compile-time constants.

Verification of an optimization. As an Alive optimization is parametric over types, the Alive interpreter checks the correctness of the optimization for each feasible type. Alive models various kinds of undefined behavior in LLVM (i.e., poison values, undef values, and true undefined behavior) [29]. The semantics of undefinedness constructs in LLVM is still being debated and will likely evolve in the future [28]. We describe verification of an optimization by excluding all kinds of undefined behavior using a definedness constraint for an instruction.

To verify an optimization for a feasible type, the Alive interpreter creates the following constraints (SMT expressions in bitvector theory) for each instruction in both the source

```

pre  ::= pred | ¬pre | pre ∧ pre | pre ∨ pre
pred ::= binpred | pfun
binpred ::= cexpr cond cexpr
cexpr  ::= constant | unop cexpr |
           cexpr binop cexpr | cfun
cond   ::= eq | ne | ugt | uge | ult |
           ule | sgt | sge | slt | sle
binop  ::= add | sub | mul | udiv | sdiv |
           urem | srem | shl | lshr | ashr |
           and | or | xor
unop   ::= neg | not
cfun   ::= abs cexpr | log2 cexpr | width value
pfun   ::= isSignBit cexpr | isPowerOf2 cexpr |
           isPowerOf2OrZero cexpr

```

Figure 1: Abstract syntax of preconditions.

and the target: (1) the expression ι that represents the result of the instruction, (2) the expression δ that represents constraints for the instruction to have defined behavior. The definedness constraints propagate with data dependencies. The interpreter also generates SMT expressions corresponding to the precondition (ϕ). A transformation is correct if and only if the target is defined and source DAG and the target DAG produce the same value for the root when the precondition is satisfied and the source is defined. That is,

$$\forall \mathcal{I} : \phi \wedge \delta_s \implies \delta_t \wedge \iota_s = \iota_t$$

where \mathcal{I} is the set of input variables in the DAG, δ_s and δ_t represent the constraints for the source and the target to be well-defined respectively, ι_s is the value computed by the source, and ι_t is the value computed by the target.

3. Precondition Inference

A precondition for a peephole optimization in LLVM is checked at compile time before applying the optimization. Hence, preconditions for these optimizations primarily deal with values that can be determined at compile time. Figure 1 provides the abstract syntax of preconditions for LLVM peephole optimizations. A precondition is a conjunction or disjunction of various predicates. A predicate is either a predicate function or a binary comparison operation involving constant expressions. Constant expressions can be symbolic constants, constant functions, and binary operations of constant expressions. There are three primary goals in designing a precondition for a peephole optimization: (1) optimization should be valid whenever the precondition is satisfied, (2) it should be as weak as possible to allow the optimization to be applicable in many scenarios, and (3) precondition should be succinct because the precondition is checked during compilation. A long precondition can increase compilation time.

Our goal is to develop an approach that can provide assistance to LLVM developers when they are adding new opti-

```

function INFERPRECONDITION( $opt, I$ )
   $\langle E^+, E^- \rangle \leftarrow \text{MAKEEXAMPLES}(opt)$ 
   $P_{valid} \leftarrow \emptyset$ 
  repeat
     $\langle P_p, P_f \rangle \leftarrow \text{PRECONDITIONSBYEXAMPLES}(E^+, E^-, I)$ 
     $e^- \leftarrow \emptyset$ 
    for all  $p \in P_p$  do
       $e_p^- \leftarrow \text{COUNTEREXAMPLES}(p, opt)$ 
      if  $e_p^- = \emptyset$  then
         $P_{valid} \leftarrow P_{valid} \cup \{p\}$ 
       $e^- \leftarrow e^- \cup e_p^-$ 
     $e_f^- \leftarrow \text{COUNTEREXAMPLES}(P_f, opt)$ 
     $e^- \leftarrow e^- \cup e_f^-$ 
     $e^+ \leftarrow \emptyset$ 
    if  $e_f^- = \emptyset$  then
       $P_{valid} \leftarrow P_{valid} \cup P_f$ 
       $e^+ \leftarrow \text{POSITIVEEXAMPLES}(P_f, opt)$ 
     $E^+ \leftarrow E^+ \cup e^+$ 
     $E^- \leftarrow E^- \cup e^-$ 
  until  $e^- = e^+ = \emptyset$ 
  return  $P_{valid}$ 

```

Figure 2: Algorithm to generate precondition for a LLVM peephole optimization opt with an initial set of predicates I . We generate an initial set of examples with the function `MAKEEXAMPLES`. The function `PRECONDITIONSBYEXAMPLES` enumerates predicates on-demand and returns a tuple: (a set of partial preconditions and a complete precondition for the sample). Both the partial preconditions and the complete precondition are checked for validity and counter examples are added to the set of bad examples. If the complete precondition is valid, it checks if it is the weakest.

mizations. Given that new peephole optimizations in LLVM are checked with the Alive tool for correctness [29], we describe our approach in the context of an Alive optimization.

High level sketch of our approach. Inspired by prior data-driven approaches [15, 37, 41, 45], we propose a data-driven approach to generate preconditions for LLVM peephole optimizations expressed in Alive. Figure 2 provides a high level sketch of our approach. Given an optimization and an initial set of predicates (which can be empty), our approach to generate a precondition consists of three main tasks. First, we generate a set of concrete positive and negative examples for the optimization. Second, we enumerate and learn predicates on-demand to separate out the positive and negative examples. When it is possible to separate out the positive and negative examples in our sample, we learn a boolean formula that eliminates all the negative examples and accepts as many positives examples as possible. Third, we check the validity of the precondition generated. If the optimization is not valid, we add the counterexamples to the set of bad examples and repeat the process. In contrast if the optimization is valid, and there are positive examples that are disallowed by the precondition, we report the partial precondition to the developer and add the concrete case to the set

of positive examples and repeat the process to weaken the precondition. Next, we describe each of these steps in detail.

3.1 Generating Examples

As we propose a data-driven precondition inference approach, we need to generate positive and negative examples for an optimization. As described earlier in Section 2, the precondition of a peephole optimization in LLVM primarily involves values that are compile time constants. Among the Alive variables, the values of symbolic constants and type parameters are available at compile time. We call them compile-time variables \mathcal{C} and denote the rest as runtime variables \mathcal{R} . Hence, our examples will have concrete valuations for the symbolic constants and the type parameters.

We call an assignment of types to type parameters and values to symbolic constants an *instance*. A single Alive optimization may have infinitely many distinct instances. Many optimizations are valid for all instances, which have an implicit precondition of `true`. An explicit precondition reduces the domain of instances where the optimization can be applied. The precondition must be false for all invalid instances, and ideally will accept as many valid instances as possible.

Positive and negative examples. Rather than examining all possible instances for an optimization, `PIInfer` works with a finite set of instances, called *examples*. An example is *positive* if, for every assignment of input variables (runtime variables) and a concrete valuation for the symbolic constants and the type parameters, the target refines the source. Otherwise, the example is *negative*.

Compile time safety of an example. While checking refinement, the Alive interpreter checks that the optimization produces defined behavior for all compile time and runtime variables as described in Section 2. When we generate examples and subsequently learn a precondition, we need to ensure that the precondition is safe (*i.e.*, it does not crash the compiler at compile time). A precondition such as $C1/C2$ can cause the compiler to crash when $C2$ is zero. In essence, a safe precondition eliminates undefined behavior among compile time variables. We will write σ for constraints to ensure compile-time safety.

From the background in Section 2, we know an optimization is correct when the refinement condition holds for all valuations of the runtime and compile time variables. We add safety of the target (σ_t) to the refinement condition because we want the generated precondition to imply safety of the target (both compile time and run-time):

$$V = \sigma_t \wedge (\delta_s \implies \delta_t \wedge \iota_s = \iota_t), \quad (1)$$

where $\iota_s, \iota_t, \delta_s, \delta_t$ and σ_t are constraints to represent the value produced by source, value produced by the target, definedness constraints for the source, definedness conditions for the target, and safety conditions for the compile time constant expressions in the target, respectively. We only con-

sider safety conditions for the target because target computes new constant expressions and the source only binds constants to variables. Hence, the compile time variables in the source are safe.

Classifying an example as positive. We classify an instance I to be positive if $V[C/I]$ is valid for all run-time variables \mathcal{R} where $V[C/I]$ is the resulting formula obtained by substituting all compile time variables in the refinement check V with concrete valuations from I . We use an SMT solver to check the validity of the above formula. **PIInfer** also automatically excludes trivial instances where the source has undefined behavior for all runtime inputs.

Providing assumptions. In addition to classifying examples as positive or negative, users may wish to influence the inference process by excluding certain examples. **PIInfer** allows the user to declare certain assumptions, such as $C \neq 0$, either because they will be ensured by other optimizations, or to focus the inference process. Given a set of predicates as assumptions A , we define $F = A \wedge \delta_s$ and consider only examples I where $F[C/I]$ is satisfiable.

Generation of examples. One challenge in generating examples is handling type variables because Alive optimizations are parametric over types. Considering only one type (e.g., largest feasible type) will likely not generate preconditions that involve constant functions like `width(%a)`, which depend on the type of the operand. Further, sampling types also avoids accidentally relying on math-based optimizations that is valid in one type but not in others. The number of type assignments are exponential in the number of type variables. Hence, we first sample the set of type assignments.

Since optimizations can have different number of type variables, we cannot create a fixed size sample of typing assignments. We increase the sample size logarithmically with the number of possible assignments, which grows exponentially with the number of type variables.

For each type assignment, we obtain examples using three methods. First, we generate random instances, by choosing assignments for each variable in \mathcal{C} . Each example is classified using F and V described earlier. Second, we generate corner cases, by creating the set $\{0, 1, -1, m\}$ for each variable in \mathcal{C} (m is the minimum signed value) and taking their Cartesian product. This ensures that certain corner cases will be included in the example set, provided they satisfy F .

Finally, we generate positive and negative examples directly, using the solver. Some optimizations are valid or invalid for a large proportion of instances, so this ensures we have both positive and negative examples. To find positive examples, we use models from the query $F \wedge \forall \mathcal{R}(V)$. Generating positive examples can be expensive as it involves a for all check with the runtime variables. To find negative examples, we use the SMT solver to find instances satisfying $A \wedge \neg V$.

```

function PRECONDITIONSBYEXAMPLES( $E^+, E^-, I$ )
   $P \leftarrow I$ 
   $M \leftarrow \text{EMPTYPREDICATEMATRIX}$ 
  for all  $p \in I$  do
     $M \leftarrow \text{ADDPREDICATE}(p, M)$ 
   $\Phi \leftarrow \emptyset$ 
  while  $\text{MIXEDVECTORS}(M) \neq \emptyset$  do
     $V_w^+, V^- \leftarrow \text{WEIGHTEDPARTITION}(M)$ 
     $\phi \leftarrow \text{LEARNPARTIALBOOLEAN}(P, V_w^+, V^-, 1)$ 
     $\Phi \leftarrow \Phi \cup \{\phi\}$ 

    Select  $v \in \text{MIXEDVECTORS}(M)$ 
     $e^+, e^- \leftarrow \text{SAMPLE}(v, M)$ 
     $p \leftarrow \text{LEARNPREDICATE}(e^+, e^-)$ 
     $P \leftarrow P \cup \{p\}$ 
     $M \leftarrow \text{ADDPREDICATE}(p, M)$ 
   $V^+, V^- \leftarrow \text{PARTITION}(M)$ 
   $\phi_f \leftarrow \text{LEARNCOMPLETEBOOLEAN}(P, V^+, V^-)$ 
  return  $\langle \Phi, \phi_f \rangle$ 

```

Figure 3: Algorithm to learn preconditions given a set of examples and an initial set of predicates (I). Function **ADDPREDICATE** adds a predicate to the predicate matrix. Function **WEIGHTEDPARTITION** partitions the predicate vectors into positive vectors and negative vectors and the weight of the positive vector is the number of positive examples accepted by the positive vector. Function **LEARNPARTIALBOOLEAN** computes the partial precondition using the weighted positive vectors and negative vectors (see Figure 5). When the predicate matrix does not have any mixed vectors, the weakest precondition is computed by the function **LEARNCOMPLETEBOOLEAN** (see Figure 7). Algorithm returns a tuple — a set of valid partial preconditions and the weakest precondition — for the given set of examples.

3.2 On-demand Predicate Enumeration and Learning

Given a set of positive and negative examples, our algorithm creates a sample of positive and negative examples from this set. It enumerates predicates on-demand until it finds a predicate that either accepts all positive examples and does not accept any negative example in the sample or accepts all negative examples and does not accept any positive example in the sample. When it finds such a predicate, it tests the predicate on the entire set of examples. When it has accumulated a set of predicates that accept all positive examples and reject all negative examples from the set of examples, it learns a Boolean formula for the precondition. Figure 3 provides a sketch of the algorithm for learning predicates from examples. Inspired by PIE [37], we separate the process of predicate enumeration and learning from the process of learning a Boolean formula. Next, we describe the algorithms in detail.

Constructing the predicate matrix. To identify whether the algorithm has learned a sufficient number of predicates to accept all positive examples and reject all negative examples, it conceptually constructs a predicate matrix. The rows in the predicate matrix correspond to the examples and the

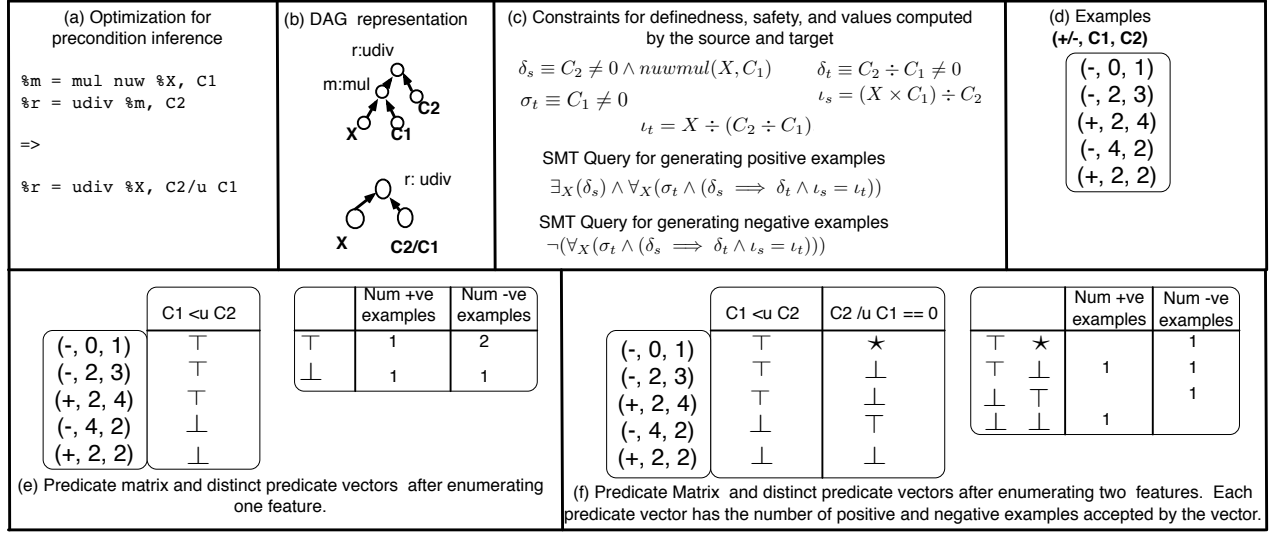


Figure 4: The process of learning preconditions. (a) LLVM peephole optimization expressed in Alive whose precondition is being learned. (b) DAG representation of the optimization, which has input runtime variable X and symbolic constants C_1 and C_2 . (c) Constraints for the definedness of the source (δ_s), definedness of the target (δ_t), compile time safety of the target (σ_t), value produced by the source (ι_s), and value produced by the target (ι_t). Queries provided to SMT solvers to generate positive and negative examples are also provided. The predicate $\text{nuwmul}(X, C_1)$ encodes the fact that X multiplied by C_1 does not overflow for the types being verified in SMT bitvector theory. (d) Sample set of examples generated. An example $(+, 4, 2)$ represents a positive example with $C_1 = 4$ and $C_2 = 2$. Any example with $C_2 = 0$ will be discarded, as it causes undefined behavior in the source. In contrast, any example with $C_1 = 0, C_2 \neq 0$ will be marked negative, because it causes an unsafe computation in the target. (e) The predicate matrix and distinct predicate vectors after adding the feature $C_1 <_u C_2$. \top indicates that the predicate accepts the example. \perp indicates that the predicate rejects the example. \star indicate that the example is unsafe (compile-time undefined behavior). (f) Predicate matrix after adding two features. Our incomplete boolean learner will find a subset of the predicates $C_1 <_u C_2, C_1 \geq_u C_2, C_2 /_u C_1 == 0$, and $C_1 /_u C_2 \neq 0$, which accepts as many of the positive vectors as it can and rejects all the negative and mixed vectors. For this matrix, it will produce $(C_1 \geq_u C_2) \wedge (C_2 /_u C_1 \neq 0)$ as the precondition.

columns correspond to the current set of predicates explored. The predicate matrix is updated whenever a new predicate is selected for separating the examples. Figure 4(e) and Figure 4(f) provide the predicate matrix with one and two predicates, respectively. Each entry in the matrix is accept (\top), reject (\perp), or unsafe (\star). A predicate accepts the example if it evaluates to true when the symbolic constants in the predicate are substituted with concrete constants from the example. Similarly, the predicate rejects an example when it evaluates to false on substitution. A predicate is unsafe with an example when it causes compile time undefined behavior when the predicate is evaluated with the concrete valuations from the example.

In Figure 4(e), the predicate $C_1 <_u C_2$ accepts the negative example $(-, 0, 1)$ and rejects the negative example $(-, 4, 2)$. Here, $<_u$ is the unsigned comparison operation. In Figure 4(f), the predicate $C_2 /_u C_1 == 0$ is unsafe with the example $(-, 0, 1)$ because it causes compile time undefined behavior (division by zero).

Predicate vectors. For efficiency, we group the rows of the predicate matrix that have the same valuation with each predicate. We call them predicate vectors. We count

the number of positive and negative examples associated with each vector. We call a predicate vector positive if it accepts only positive examples. Similarly, a predicate vector is negative if it accepts only negative examples. We call the predicate vector a mixed vector if it accepts both positive and negative examples.

Partial preconditions. If there is a positive predicate vector, then a partial precondition is possible if it rules out all negative examples. Of course, it would accept a subset of the positive examples. When there are positive vectors, our algorithm learns the partial precondition using the partial Boolean formula learning algorithm in Figure 5. Any mixed vector is treated as a negative vector while generating partial preconditions. Each positive vector is weighted by the number of positive examples that it accepts. Although the partial precondition generated from the formula learner is valid for the set of examples, it can be invalid for the Alive optimization. All partial preconditions are checked for validity before reporting it to the compiler developer as shown in Figure 2.

When there are no mixed vectors, we have generated a set of predicates that separate the positive and negative examples in consideration. The complete Boolean learner at-

tempts to learn the weakest precondition. As with the partial precondition, the complete precondition is checked for validity for the Alive optimization. The process is repeated by adding counterexamples if it is invalid or by adding positive examples if it is not the weakest possible precondition.

Predicate enumeration. When there are mixed vectors, the current set of predicates is not sufficient to completely separate the positive and negative examples. Initially, an empty set of predicates is a mixed vector as it accepts both positive and negative examples. Our algorithm enumerates new predicates when there are mixed vectors. The goal of this phase is to generate a new predicate that splits a mixed vector into either a positive vector or a negative vector.

We generate predicates using bounded recursion. Each predicate is assigned a weight, and we enumerate predicates with increasing weights. The weight of a predicate roughly corresponds to the number of leaf nodes in the abstract syntax tree (AST) of the predicate in Alive’s internal representation. Exceptions are predicate functions and constant functions, which contribute to the weight but are not leaf nodes. Symbolic and literal constants have a weight of 1. The weight of the predicate and constant functions are one more than the total weight of their arguments. The weight of a comparison predicate is equal to the sum of the weights of the sub-expressions. The minimum weight of a predicate is two. To generate an expression of weight w , our enumerator recursively generates two subexpressions of weight $w1$ and weight $w2$ such that $w = w1 + w2$.

Type polymorphism in the predicate language also introduces additional challenges in enumeration. For example, comparisons require their arguments to have the same type. Hence, enumeration is also type-directed. To avoid generating ill-typed expressions, our expression enumerator takes the target type as a parameter when generating constant expressions. It chooses appropriate symbols with the right type when it reaches the leaf nodes. To avoid infinite recursion, we can require that the argument to a unary operator must not be another unary operator expression, which is necessary because the unary operator does not increase the weight of the expression.

To avoid generating equivalent expressions (e.g., $a + (b + c)$, $(a + b) + c$, $(b + a) + c$), our enumerator is aware of the algebraic properties of the predicate language, and produces expressions in a normal form. However, we have to be careful in applying only algebraic identities with bitvector arithmetic. For example, $-(a \div b)$, $-a \div b$, and $a \div -b$ are all distinct expressions with bitvector arithmetic.

Testing the enumerated predicate on a sample. Typically, our algorithm enumerates a large number of predicates before we discover a predicate that splits the mixed vector into positive and a negative vector. We test the enumerated predicate on a subset of the examples accepted by the mixed vector. Testing the enumerated predicate on a small sample quickly rules many of the enumerated predicates when com-

```

function LEARNPARTIALBOOLEAN( $Preds, V_w^+, V^-, K$ )
   $lits \leftarrow Preds \cup \{\neg p : p \in Preds\}$ 
   $D \leftarrow \{\bigvee d : d \in \mathcal{P}(lits), |d| \leq K\}$ 
   $C \leftarrow \emptyset$ 
  while  $\exists v \in V^-$  s.t.  $ACCEPTS(\bigwedge C, v)$  do
     $A \leftarrow \{\langle w, v \rangle : \langle w, v \rangle \in V_w^+, ACCEPTS(\bigwedge C, v)\}$ 
     $c \leftarrow \operatorname{argmax}_{d \in D} \sum \{w : \langle w, v \rangle \in A, ACCEPTS(d, v)\}$ 
     $C \leftarrow C \cup \{c\}$ 
     $D \leftarrow D \setminus \{c\}$ 
    if  $D = \emptyset$  then
      return  $\perp$ 
  return COVERCLAUSES( $C, V^-$ )

```

Figure 5: Algorithm to learn a partial Boolean that rejects all negative vectors and maximizes the weights of the positive vectors accepted. $\mathcal{P}(lits)$ represents the power set of literals.

pared to testing the predicate on the entire collection of examples.

We enumerate predicates until we find one that divides the sample: either accepting all positive examples and no negative examples, or rejecting all positive examples and no negative examples. Predicates may be unsafe for negative examples, but cannot be unsafe for any positive examples. When a predicate is selected, it is tested against all examples, and discarded if is unsafe for any positive example. Although the learned predicate separates the sample, it may not completely separate the positive and negative examples in the complete set of examples. Yet, such a predicate can be helpful in narrowing the search for future predicates.

Our algorithm subsequently regenerates new predicate vectors as a result of adding the new feature to the predicate matrix. The entire process is repeated until there are no mixed vectors. We perform Boolean formula learning to generate a precondition when there are no mixed vectors (weakest precondition) or when there exists at least one positive vector (partial precondition). Any mixed vector is treated as a negative vector while generating partial preconditions.

3.3 Boolean Formula Learning with Weighted Vectors

This phase of our framework learns a Boolean formula in conjunctive normal form (CNF) using the set of predicates that separate the positive and negative examples. Compiler writers typically want succinct preconditions, which have fewer clauses than the weakest possible precondition, because each of these clauses have to be checked at compile time. The weakest precondition for some optimizations can be complex. Succinct preconditions that reject some positive examples are also preferred. Hence, we generate both a set of partial preconditions that are valid for the optimization along with the weakest precondition.

To generate both succinct/partial and weakest preconditions, we propose two Boolean learning algorithms: a partial and a complete Boolean formula learner. The key idea

```

function COVERCLAUSES( $C, V^-$ )
   $P \leftarrow \top$ 
  while  $\exists v \in V^-$  s.t. ACCEPTS( $P, v$ ) do
     $c \leftarrow \operatorname{argmax}_{d \in C} |\{v : v \in V^-, \neg \text{ACCEPTS}(d, v)\}|$ 
     $V^- \leftarrow V^- \setminus \{v : v \in V^-, \neg \text{ACCEPTS}(c, v)\}$ 
     $C \leftarrow C \setminus \{c\}$ 
     $P \leftarrow P \wedge c$ 
  return  $P$ 

```

Figure 6: Greedy set cover algorithm that returns a set of clauses that cover all negative examples.

in these formula learners is to find a formula that rejects all negative examples while accepting all positive examples (for the weakest precondition) or as many positive examples as possible (for partial preconditions).

Weighted Partial Boolean formula learner. The partial Boolean learner takes the set of predicates, the set of weighted positive vectors where the weight for each vector is the number of positive examples associated with that vector, and the set of negative vectors as input. Figure 5 provides the algorithm for the weighted partial Boolean formula learner. Its goal is to generate a succinct precondition, which necessarily may not be the weakest possible precondition. Hence, it generates all possible clauses up to size K , where K is typically a small number (one or two). Among these clauses, the goal is to select enough clauses to reject all negative examples while maximizing the amount of positive examples accepted. The algorithm starts with an empty set of chosen clauses. When there exists a negative vector which is accepted by all the current set of chosen clauses, it selects a clause that maximizes the number of positive examples accepted. After the clause is chosen, the positive vectors rejected by the clause is not available for selection in the subsequent steps. It recomputes the available vectors (A in Algorithm 5). As the algorithm at the end generates a CNF formula, all the positive vectors have to be accepted by all the chosen clauses. Finally, the algorithm computes a minimal set of clauses that is sufficient to cover all the negative examples with an approximate set cover algorithm in Figure 6. We have a two step process because the minimal cover algorithm selects from a set of chosen clauses but it does not know which ones to select until it gets them all. Figure 8(a–c) illustrates the partial boolean learner that maximizes the weight of the positive vectors accepted while generating succinct preconditions.

Complete Boolean formula learner. Algorithm 7 describes our Boolean formula learner, which generates a formula that accepts all positive vectors and rejects all negative vectors. Initially the set of chosen clauses is empty. While there exists a negative vector that is accepted by the current set of chosen clauses, it enumerates all K -CNF clauses of length K . It adds a clause to the set of chosen clauses if the clause accepts all positive vectors. It iteratively gener-

```

function LEARNCOMPLETEBOOLEAN( $Preds, V^+, V^-$ )
   $lits \leftarrow Preds \cup \{\neg p : p \in Preds\}$ 
   $k \leftarrow 0$ 
   $C \leftarrow \emptyset$ 
  while  $\exists v \in V^-$  s.t. ACCEPTS( $\bigwedge C, v$ ) do
     $k \leftarrow k + 1$ 
     $C_k \leftarrow \{\bigvee d : d \in \mathcal{P}(lits), |d| = k\}$ 
     $C \leftarrow C \cup \{d : d \in C_k, \forall v \in V^+ (\text{ACCEPTS}(d, v))\}$ 
  return COVERCLAUSES( $C, V^-$ )

```

Figure 7: Learn a Boolean formula given a set of predicates, positive vectors, and negative vectors.

ates clauses of increasing complexity and repeats the above process. Finally, once all the negative vectors are rejected, it uses the approximate set cover algorithm (see Algorithm 6) to generate a minimum set of clauses to reject all negative examples. The complete boolean learner generates the weakest precondition for the given set of examples. Figure 8(d–e) illustrates the complete boolean learner that accepts all positive vectors and rejects all negative vectors.

4. Evaluation

We describe the PInfer prototype, our methodology, and our experience inferring preconditions for LLVM peephole optimizations. Our experiments evaluate the effectiveness of the PInfer prototype in generating partial and the weakest possible preconditions.

The PInfer prototype. We built the PInfer prototype by extending the publicly available Alive-NJ source code. Alive has experimental support for memory-related optimizations, `getelementptr`, and control-flow optimizations. Alive-NJ also supports floating point optimizations. We exclude them from the PInfer prototype.

PInfer enhances Alive-NJ with these major features. (1) Implementations of the predicate enumerator and precondition inference algorithms, comprising roughly two thousand lines of Python code. (2) A safety analysis, which expresses the conditions under which an optimization target or precondition may have undefined behavior at compile-time. (3) A separation of Alive-NJ’s type checking and type assignment phases. PInfer assigns each term an abstract type once during type checking or predicate enumeration. These abstract types are then mapped to concrete types during validation without the need of re-performing type checking.

In our experiments, we use Z3 4.4.1 [10] to handle SMT queries. The PInfer prototype is open source and publicly available.¹

Optimization suite. Alive is distributed with a suite of 417 optimizations, a snapshot of optimizations from LLVM’s InstCombine and InstructionSimplify passes. Some

¹incompletes branch of the repository at <https://github.com/rutgers-apl/alive-nj>.

V^+	w	p_1	$\neg p_1$	p_2	$\neg p_2$	p_3	$\neg p_3$
$\perp\perp\perp$	8		+	+			+
$\perp\perp\top$	8		+	+		+	
$\top\perp\perp$	10	+			+		+
$\top\perp\top$	1	+			+	+	
$\top\top\top$	3	+		+		+	
		14	16	19	11	12	18
V^-							
$\perp\perp\perp$		-		-		-	
$\perp\perp\top$		-		-		-	
$\top\perp\perp$			-		-	-	
$\top\top\perp$				-		-	

(a) Select p_2 , discard $\top\perp\perp$, $\top\perp\top$, $\perp\perp\perp$ and $\perp\perp\top$.

V^+	w	p_1	$\neg p_1$	p_2	$\neg p_2$	p_3	$\neg p_3$
$\perp\perp\perp$	8		+	+			+
$\perp\perp\top$	8		+	+		+	
$\top\top\top$	3	+		+		+	
		3	16		0	11	8
V^-							
$\top\top\perp$			-		-	-	

(b) Select $\neg p_1$, discard $\top\top\top$ and $\top\top\perp$.

V^+	w	p_1	$\neg p_1$	p_2	$\neg p_2$	p_3	$\neg p_3$
$\perp\perp\perp$	8		+	+			+
$\perp\perp\top$	8		+	+		+	
		0		0	8	8	
V^-							

(c) Final: $p_2 \wedge \neg p_1$.

V^-	$p_1 \vee p_2$
$\perp\perp\perp$	-
$\perp\perp\top$	-
$\top\top\perp$	-

(d) 2-CNF terms

V^-	$p_1 \vee p_2$	$p_1 \vee p_2 \vee p_3$	$p_1 \vee p_2 \vee \neg p_3$	$\neg p_1 \vee \neg p_2 \vee p_3$
$\perp\perp\perp$	-	-		
$\perp\perp\top$	-		-	
$\top\top\perp$				-

(e) 3-CNF terms. Cover is $(p_1 \vee p_2) \wedge (\neg p_1 \vee \neg p_2 \vee p_3)$.

Figure 8: Illustration of the partial and complete Boolean learners on the same predicate matrix. In each table, rows correspond to vectors and columns correspond to clauses. A + indicates that the clause accepts a positive vector, and a - indicates that it rejects a negative vector. (a–c) show how the partial learner selects clauses until all negative clauses are rejected. In (a), the algorithm selects p_2 as it maximizes the weight and it discards positive vectors $\top\perp\perp$, $\top\perp\top$, $\perp\perp\perp$, and $\perp\perp\top$ because p_2 rejects them. (d–e) show how the complete learner adds larger clauses until all negative clauses are rejected. Any clause considered by the complete learner has to accept all positive vectors. Only clauses which accept all positive vectors are shown in this figure.

preconditions in Alive are weaker than LLVM’s preconditions. Of these 417 optimizations, 195 require no precondition and 41 rely on run-time analyses. Of the remaining 181, seven require constant functions or predicates not currently supported by PlInfer. This leaves us with 174 optimizations for which PlInfer could possibly derive preconditions.

Methodology. In our experiments for precondition inference, we removed the precondition in the optimization and provided it to the PlInfer prototype. We compare the precondition generated by the PlInfer prototype and original precondition for it in Alive (to determine if it is weaker or stronger). All experiments were performed on a 64-bit Intel Skylake processor machine with four cores and 16 GB of RAM.

Precondition enumerator vs predicate learner. To perform a fair comparison of the benefits of predicate learning, we modified PlInfer to enumerate all possible preconditions and test each one for validity and completeness. We call it precondition enumerator. We produce preconditions as combinations of one or more predicates along with the logical connectives and, or, and not. To avoid generating too many equivalent forms, preconditions are generated in conjunctive normal form, *i.e.* a conjunction of one or more disjunctions of one or more possibly-negated predicates. The size of a precondition is the sum of the sizes of the predicates it contains.

This algorithm obtains preconditions in non-decreasing order of size. Each precondition is tested against a set of example instances generated in the same manner as the predicate learner. If the precondition accepts all positive instances and does not accept any negative instances, it is then verified by the SMT solver. If the solver finds counter examples, or additional positive examples which the precondition rejects,

testing continues with the next precondition. It is not necessary to reconsider a previously-rejected precondition.

This algorithm has two advantages over predicate learning: it always finds a precondition of minimum size, and it can never get stuck with a bad set of examples.² The disadvantage is that this algorithm cannot break the search problem into smaller parts. The numbers of preconditions and predicates for a given size both grow exponentially, with the number of preconditions growing somewhat faster. Consider an optimization for which the minimally-sized precondition has two predicates of sizes m and n . The number of preconditions which must be searched will be $O(c^{m+n})$, which is vastly larger than $O(c^m + c^n)$, the best-case amount of work needed for the predicate learner to find the two predicates. The exponential growth means that the predicate learner is still faster even if it wastes most of its effort learning predicates which will later be discarded.

We will refer to our on-demand predicate learner as Infer and our precondition enumerator as Search in the rest of the evaluation.

Effectiveness in generating preconditions. We test the effectiveness of our approach by generating preconditions for the optimizations in the Alive suite. We ran inference for each optimization with a 1000-second timeout. Our on-demand predicate learner successfully generated the weakest precondition for 133 out of the 174 optimizations. Although PlInfer could not generate the weakest possible precondition for the remaining 41 optimizations, it generated a partial precondition using the weighted partial Boolean learning algorithm for 31 optimizations. Six optimizations did not

² A bad set, in this case, would be one which is divided too easily, leading to extra work for the solver.

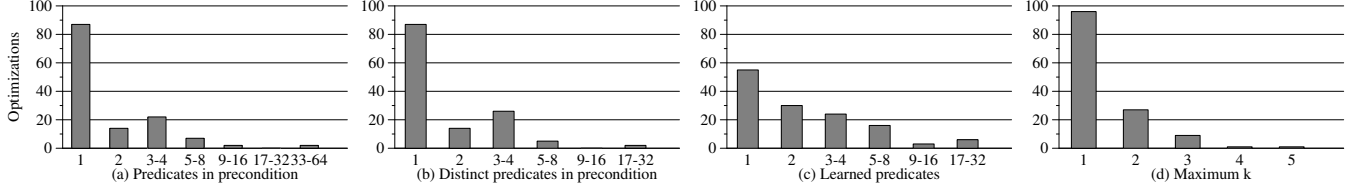


Figure 9: Information about the weakest preconditions successfully inferred within 1000 s. The histograms show the number of optimizations with (a) the number of predicates in the precondition, (b) the number of distinct predicates in the precondition (a predicate and its negation are not considered distinct), (c) the number of predicates accepted by the learner during inference, and (d) the maximum number of predicates occurring in a disjunction (*i.e.*, the value of k reached by the Boolean formula learner).

generate either a partial or a complete precondition within the timeout period because it could not find a valid predicate that separates positive and negative examples. Our Boolean learner could not generate a formula that rejects all negative vectors for one optimization. The PInfer prototype could not generate any precondition for three optimizations because Z3 returned unknown either in example generation or in final validation. In summary, on-demand predicate learner was able to generate either the weakest possible precondition or a partial precondition for 165/174 optimizations.

Figure 9 provides summarized information on the number of predicates in the generated weakest precondition, number of distinct predicates in the weakest precondition, number of features accepted by the predicate learner to separate out positive and negative examples, and maximum disjunction size in the final formula learned by the Boolean learner. About 80 optimizations in the suite have a single predicate in the precondition (see Figure 9(a)). Around 40 optimizations have more than 1 and up to 4 predicates (see Figure 9(a)). Figure 9(b) shows that the number of distinct predicates in the weakest precondition is lower than the number of predicates. This is common in formulas expressed in conjunctive normal form.

Figure 9(c) reveals that the on-demand learner generates up to 32 predicates to separate the examples, which is simplified to a small number of predicates in the final precondition by the Boolean learner. There are 45 optimizations with a clause size of 2 or more in the weakest precondition generated by the predicate learner (see Figure 9(d)).

The on-demand learner enumerated more than 10K predicates for 9 optimizations (highest being 104,000 predicates), between 1K–10K predicates for 19 optimizations, 100–1K for 16 optimizations, 11–100 predicates for 86 optimizations, and 1–10 predicates for 4 optimizations. We report this data only for optimizations for which the learner was able to synthesize the weakest precondition.

Predicate learner versus precondition enumerator. For comparison, we also experimented with precondition enumerator (described earlier in this section) with same set of optimizations. In contrast to on-demand predicate learner, precondition enumerator was able to generate the weakest precondition for 114/174 optimizations. It performs similar

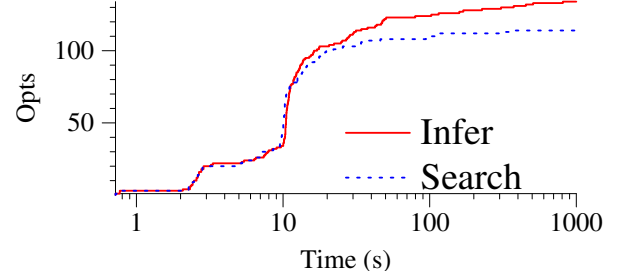


Figure 10: Number of preconditions which can be inferred by the on-demand predicate learner (Infer) and the precondition enumerator (Search) within a given time limit. The x -axis is running time, in seconds. The y -axis is the number of optimizations for which the precondition can be inferred in that time.

to on-demand predicate learner for optimizations with few predicates in the precondition. It times out for optimizations that have more than three predicates in the precondition.

Figure 10 reports the number of optimizations for which the on-demand predicate learner and the precondition enumerator are able to generate the weakest precondition within a given amount of time. The on-demand predicate learner and the precondition enumerator both generate preconditions for around 100 optimizations in less than 10 seconds, which correspond to optimizations with one or two predicates in the precondition (see Figure 9(a)). In summary, we observe that the on-demand learner can infer preconditions for more optimizations than the precondition enumerator within a given time limit.

Comparison with LLVM’s/Alive Precondition. We compare the weakest possible precondition generated by PInfer and the precondition in the Alive suite. We consider a precondition generated by PInfer to be the weakest possible precondition if it accepts every instance where the optimization is non-trivially valid; *i.e.*, the source is well-defined for some run-time input. To determine if the precondition generated by PInfer (ϕ_{infer}) is weaker than LLVM’s (ϕ_{llvm}), we check the satisfiability of the formula — $(\phi_{infer} \wedge \neg \phi_{llvm})$ — using the SMT solver.

Among the 133 optimizations where PInfer was able to generate the weakest possible precondition, 73 were weaker than LLVM’s precondition. Figure 11 provides a sample

<p>(1) Select:423</p> <pre> %and = and %X, C1 %c = icmp eq %and, 0 %F = and %X, C2 %r = select %c, %X, %F => %r = and %X, C2 LLVM Precondition: isPowerOf2(C1) && C1 == -C2 PInfer Precondition: (-C1 & -C2) == 0 </pre>	<p>(2) AndOrXor:922</p> <pre> %op0 = icmp eq %a, C1 %op1 = icmp ne %a, C2 %r = and %op0, %op1 => %r = icmp eq %a, C1 LLVM Precondition: C1 u< C2 PInfer Precondition: C1 != C2 </pre>	<p>(3) AndOrXor:363</p> <pre> %lhs = or %A, C1 %Op = add %lhs, %B %r = and %Op, C2 => %op = add %A, %B %r = and %op, C2 LLVM Precondition: isPowerOf2OrZero(C2+1) && C1 & C2 == 0 PInfer Precondition: C1 - 1 u>= C2 && (C2 & C1) == 0 && (C1 == 0 isPowerOf2(C1) (-C1 ^ C1) + C2 < 0) </pre>	<p>(4) AndOrXor:210</p> <pre> %op = shl %X, C1 %r = and %op, C2 => %r = and %op, C2 & (-1 << C1) LLVM Precondition: (C2 & (-1 << C1)) != -1 << C1 PInfer Precondition: width(%r) u> C1 </pre>
---	---	--	---

Figure 11: A sample of optimizations where PInfer generated a weaker precondition compared to the precondition in LLVM/Alive. We provide the name of the optimization in the Alive suite, the LLVM/Alive precondition and the PInfer precondition. (1) Consider the instance $C1 = 3$, $C2 = 14$ for 4-bit integers, *i.e.*, 0011 and 1110. These satisfy neither of the clauses in LLVM’s precondition, but do satisfy the PInfer’s precondition, which can be rewritten as $C1 \mid C2 == -1$. (2) This optimization’s source calculates $a = C1 \wedge a \neq C2$ and the target $a = C1$. By the transitive property, this is equivalent to $a = C1 \wedge C1 \neq C2$. PInfer generates the equivalent precondition $C1 > C2 \mid C1 < C2$. (3) Consider the instance $C1 = 10$, $C2 = 2$ for 4-bit integers, *i.e.*, 1100 and 0010. This is rejected by LLVM’s precondition, because three is not a power of two, but is accepted by PInfer’s. (4) PInfer’s precondition is clearly weaker, as it will accept the cases where $C2$ is masked by $-1 \ll C1$ as long as $C1$ is less than the bit width. For example, $C1 = 2$, $C2 = 14$ for 4-bit integers, *i.e.*, 0001 and 1110.

of four optimizations out of the 73 where PInfer generated a weaker precondition than LLVM. For the remaining 61 optimizations, PInfer generated the same precondition as LLVM. Even the partial preconditions generated were weaker than LLVM preconditions. Among the partial precondition generated by PInfer, 15 of them were incomparable because there were instances which were accepted by the PInfer precondition but not by the LLVM precondition and vice-versa. We also generated partial precondition, which is incomparable to the LLVM precondition, for the example in Section 1. In summary, PInfer generates weaker preconditions than LLVM preconditions in the Alive suite.

One reason that PInfer generates many preconditions weaker than LLVM can be attributed to implicit assumptions in the staging of optimizations. LLVM assumes all prior optimizations in the LLVM peephole optimization suite has been attempted and foldable constants have been eliminated. PInfer learns preconditions in isolation; assumptions must be explicit.

5. Generalizing Concrete Expression DAGs

To further demonstrate the applicability of PInfer, we generalize some concrete optimization instances generated by a LLVM-IR based superoptimizer with PInfer. Souper [21, 39] is an open-source effort to develop an LLVM-IR based super optimizer. An initial version of Souper collects a database of expressions DAGs that would evaluate to either true or false [39]. It also generates concrete path conditions with such expression DAGs. We focus on expression DAGs without path conditions, which can be translated to Alive.

Figure 12(1a) and Figure 12(2a) present the expression DAGs in Souper syntax. The corresponding instance in Alive syntax is shown in Figure 12(1b) and Figure 12(2b), re-

spectively. We create a generalized version of the expression DAG by replacing all concrete constants in the source with symbolic constants. However, we cannot replace the concrete constant in the target with a symbolic constant as Alive syntax forbids defining new symbolic constants in the target. The generalized optimization and preconditions generated by PInfer are shown in Figure 12(1c) and Figure 12(2c).

We translated 71 expression DAGs from the initial results of Souper to Alive and generalized them with symbolic constants. PInfer was able to generate the weakest possible precondition for 51 optimizations. It generated partial preconditions for additional 3 optimizations. We were not able to generalize 17 optimizations because of the following reasons: (1) Z3 would hang while generating positive examples or Z3 would return unknown, and (2) PInfer could not learn a Boolean formula that rejects all negative vectors.

The weakest possible precondition generated by PInfer for the generalized optimization in Figure 12(2c) is

```

(C3 != 0 || C2 == 0) &&
(C2 u<= 1 || (C4 & ~C1) != 0 || C4 < 0) &&
((C4 & ~C1) != 0 || C4 >= C2) &&
(C3 != 0 || C1 == 0) &&
(C2 != 0 || C4 == 0 || (C4 & ~C1) != 0) &&
(C2 u> 1 || C4 u<= 1 || (C4 & ~C1) != 0) &&
(isSignBit(C4) || C2 + 1 >= 0 || (C4 & ~C1) != 0)

```

Unfortunately, the weakest precondition is not succinct. The partial precondition generated by our weighted partial Boolean formula learner is $(C4 \& \sim C1) != 0 \wedge C3 != 0$ (also shown in Figure 12(1c)). It is succinct and accepts 95% of the positive examples, which makes a case for generating partial preconditions.

(1a) Souper pattern Program 512 <pre> %0:i32 = var %1:i32 = and 1:i32, %0 %2:i1 = eq 0:i32, %1 %3:i1 = xor 1:i1, %2 %4:i1 = ne 0:i32, %1 %5:i1 = and %3, %4 %6:i1 = or %5, %2 cand %6 1:i1 </pre>	(1b) Alive translation <pre> %1 = and i32 1, %0 %2 = icmp eq 0, %1 %3 = xor 1, %2 %4 = icmp ne 0, %1 %5 = and %3, %4 %6 = or %5, %2 => %6 = 1 </pre>	(1c) Generalized optimization PIInfer precondition: <pre> ((C4 & ~C1) != 0 && C3 != 0) </pre> <pre> %1 = and i32 C1, %0 %2 = icmp eq C2, %1 %3 = xor C3, %2 %4 = icmp ne C4, %1 %5 = and %3, %4 %6 = or %5, %2 => %6 = 1 </pre>	(2a) Souper pattern Program 537 <pre> %0:i32 =var %1:i32 =srem 1:i32,%0 %2:i1 =lshr %1, 1 1:i32 %3:i1 =ne 0:i1, %2 cand %3 0:i1 </pre>	(2b) Alive translation <pre> %1 =srem i32 1, %0 %2 =lshr %1, 1 %3 =icmp ne 0, %2 => %3 = 0 </pre>	(2c) Generalized optimization PIInfer precondition: <pre> (C3 == 0 && (C1 u>> C2) == 0) </pre> <pre> %1 = srem i32 C1, %0 %2 = lshr %1, C2 %3 = icmp ne C3, %2 => %3 = 0 </pre>
---	---	--	--	--	--

Figure 12: Generalization of concrete instances generated by Souper with PIInfer. (a) Concrete expression DAG in Souper format. (b) Translated Alive representation. (c) Generalized optimization with all concrete constants in the source replaced by symbolic constants.

6. Related Work

There is a large body of work on inferring specifications — preconditions, postconditions, and invariants — for general purpose programs [1, 3, 4, 6, 9, 11–15, 37, 41, 44, 45]. Data-driven approaches have also been explored for inferring specifications [13–15, 37, 41]. We primarily focus on closely related work in this section.

PIE vs PIInfer. Our work is inspired by PIE [37], which generates preconditions for general purpose programs. PIE uses on-demand predicate learning, which it calls feature learning, along with a Boolean learner to separate positive and negative examples. PIInfer differs from PIE by addressing new challenges in an LLVM/Alive context. First, we identify the need for succinct/partial preconditions and propose a weighted partial Boolean formula learner. Second, we propose a strategy to generate positive and negative examples while handling polymorphic types and compile-time undefined behavior, which necessitates ternary values (accept, reject, unsafe) for each predicate in predicate learning. Third, we propose a strategy to enumerate predicates and compare it with precondition enumeration.

Compiler precondition synthesis. Prior approaches have also explored precondition generation for compiler optimizations [7, 30, 42]. PSyCO [30] synthesizes read-write preconditions given a finite predicate set. They do not address the complexities of bitvector arithmetic and the interaction with undefined behavior. Optgen [7] automatically generates all peephole optimizations within a specified bound and verifies their correctness. These optimizations may include preconditions, which are expressions of the form `expr == 0`, when the optimization is correct. These are found through precondition enumeration.

Logical abduction methods. Alternative methods for precondition inference include logical abduction [11, 16]. Methods using quantifier elimination [11] are promising, but methods for eliminating quantifiers in bitvector algebra work only for a small subset of operations [19]. We initially tried logical abduction methods by restricting optimizations to use only linear integer arithmetic (LIA) but settled on a data-driven approach to increase its applicability.

Data-driven inference methods. Other prior data-driven approaches often work only with predefined predicates [15, 41, 45]. Researchers have used counter-example guided refinement [8], similar to PIInfer, by beginning with overlapping positive and negative sets and refining them by finding counter-examples [44]. They also require a fixed set of predefined predicates. ICE/ICE-DT [13, 14] use positive, negative, and implication examples for synthesizing invariants. They use either a template based synthesis or a decision tree learning algorithm to generate invariants using a fixed set of attributes. PIInfer, similar to PIE, learns and synthesizes predicates on-demand.

Search techniques and super optimization. PIInfer’s inference can be viewed as a variant of various symbolic, stochastic, and/or enumerative search strategies employed in program synthesis [2, 17, 22, 46, 48] and super optimizers [5, 20, 32, 38, 43]. PIInfer is applicable in generalization/validation of patterns generated by super optimizers.

Compiler correctness. The compiler can be written in a mathematical theorem prover (*e.g.*, CompCert [26], Vellvm [50, 51]), which would require one to figure out the specification in such a setting [35, 47]. Alternatively, various other DSLs have also been proposed for compiler construction [23, 25, 31]. PIInfer generates preconditions or optimizations expressed in Alive [29]. In principle, PIInfer can apply to other DSLs. Our recent work has explored compiler non-termination errors with a suite of peephole optimizations [33], which typically occurs when profitability metrics are included in the precondition. The weakest preconditions inferred by PIInfer should be checked with those tools before including them in LLVM to avoid non-termination errors.

7. Conclusion

We show that it is possible to infer preconditions for peephole optimizations in LLVM using a data-driven approach by learning predicates on-demand. We highlight the trade-offs between applicability and succinctness of the precondition. The PIInfer prototype addresses the challenges of polymorphic types and compile-time undefined behavior in the precondition language to generate both weakest and succinct

partial preconditions. Our primary goal is to assist LLVM developers in debugging an invalid optimization. We believe PInfer is useful to LLVM developers as it is able to generate preconditions weaker than LLVM’s preconditions.

Acknowledgments

We thank Aarti Gupta for feedback on the paper. We also thank John Regehr for his blog posts on Souper and for providing concrete expression DAGs generated by Souper. This paper is based on work supported in part by NSF CAREER Award CCF-1453086, a sub-contract of NSF Award CNS-1116682, a NSF Award CNS-1441724, a Google Faculty Award, and gifts from Intel Corporation.

References

- [1] A. Albarghouthi, I. Dillig, and A. Gurfinkel. Maximal Specification Synthesis. In *Proceedings of the 43rd Annual Symposium on Principles of Programming Languages*, POPL, pages 789–801, Jan. 2016.
- [2] R. Alur, R. Bodík, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-Guided Synthesis. In *Proceedings of the 13th International Conference on Formal Methods in Computer-Aided Design*, FMCAD, pages 1–17, Oct. 2013.
- [3] R. Alur, P. Černý, P. Madhusudan, and W. Nam. Synthesis of Interface Specifications for Java Classes. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 98–109, Jan. 2005.
- [4] G. Ammons, R. Bodík, and J. R. Larus. Mining Specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 4–16, 2002.
- [5] S. Bansal and A. Aiken. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 394–403, Oct. 2006.
- [6] M. Barnett and K. R. M. Leino. Weakest-precondition of Unstructured Programs. In *Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE, pages 82–87, Sept. 2005.
- [7] S. Buchwald. Optgen: A Generator for Local Optimizations. In *Proceedings of the 24th International Conference on Compiler Construction*, CC, pages 171–189, 2015. ISBN 978-3-662-46663-6.
- [8] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. CAV, 2000.
- [9] P. Cousot, R. Cousot, M. Fähndrich, and F. Logozzo. Automatic Inference of Necessary Preconditions. In *Proceedings of the 14th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI, pages 128–148, Jan. 2013.
- [10] L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS, pages 337–340, 2008.
- [11] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive Invariant Generation via Abductive Inference. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA, pages 443–456, Oct. 2013.
- [12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1):35–45, Dec. 2007.
- [13] P. Garg, C. Löding, P. Madhusudan, and D. Neider. ICE: A Robust Learning Framework for Synthesizing Invariants. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV, pages 69–87, July 2014.
- [14] P. Garg, D. Neider, P. Madhusudan, and D. Roth. Learning Invariants using Decision Trees and Implication Counterexamples. In *Proceedings of the 43rd Annual Symposium on Principles of Programming Languages*, POPL, pages 499–512, Jan. 2016.
- [15] T. Gehr, D. Dimitrov, and M. T. Vechev. Learning Commutativity Specifications. In *Proceedings of the 27th International Conference on Computer Aided Verification*, CAV, pages 307–323, July 2015.
- [16] R. Giacobazzi. Abductive analysis of modular logic programs. In *Proceedings of the 1994 International Symposium on Logic programming*, ISPL, pages 377–391, Nov. 1994.
- [17] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of Loop-free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, June 2011.
- [18] Y. Jiang. [Patch]InstCombine pattern for ICMP. <http://lists.llvm.org/pipermail/llvm-commits/Week-of-Mon-20140818/231300.html>, 2014. Retrieved 2016-11-10.
- [19] A. K. John and S. Chakraborty. Quantifier Elimination for Linear Modular Constraints. In *Proceedings of the 4th International Congress on Mathematical Software*, ICMS, pages 295–302, Aug. 2014.
- [20] R. Joshi, G. Nelson, and Y. Zhou. Denali: A practical algorithm for generating optimal code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 28(6):967–989, Nov. 2006.
- [21] J. Ketema, J. Regehr, J. Taneja, P. Collingbourne, and R. Sasnauskas. A superoptimizer for LLVM IR. <https://github.com/google/souper>. Retrieved 2016-11-14.
- [22] A. Komuravelli, A. Gurfinkel, and S. Chaki. SMT-Based Model Checking for Recursive Programs. In *Proceedings of the 26th International Conference on Computer Aided Verification*, CAV, pages 17–34, July 2014.
- [23] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming*

- Language Design and Implementation*, PLDI, pages 327–337, 2009.
- [24] V. Le, M. Afshari, and Z. Su. Compiler Validation via Equivalence Modulo Inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 216–226, 2014.
 - [25] S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated Soundness Proofs for Dataflow Analyses and Transformations via Local Rules. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 364–377, 2005.
 - [26] X. Leroy. A Formally Verified Compiler Back-end. In *Journal of Automated Reasoning*, 2009.
 - [27] C. Liam. [Patch]Implementing a proposed InstCombine optimization. <http://lists.llvm.org/pipermail/llvm-dev/2016-April/098104.html>, 2016. Retrieved 2016-11-10.
 - [28] N. Lopes. RFC: Killing undef and spreading poison. <http://lists.llvm.org/pipermail/llvm-dev/2016-October/106182.html>, 2016. Retrieved 2016-11-10.
 - [29] N. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably Correct Peephole Optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 22–32, 2015.
 - [30] N. Lopes and J. Monteiro. Weakest Precondition Synthesis for Compiler Optimizations. In *Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI, pages 203–221, 2014.
 - [31] N. P. Lopes and J. Monteiro. Automatic Equivalence Checking of UF+IA Programs. In *Proceedings of the 20th International Symposium on Model Checking Software*, SPIN, pages 282–300, July 2013.
 - [32] H. Massalin. Superoptimizer: A Look at the Smallest Program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 122–126, 1987.
 - [33] D. Menendez and S. Nagarakatte. Termination-Checking for LLVM Peephole Optimizations. In *Proceedings of the 38th International Conference of Software Engineering*, ICSE, pages 191–202, May 2016.
 - [34] D. Menendez, S. Nagarakatte, and A. Gupta. Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM. In *Proceedings of the 23rd Static Analysis Symposium*, pages 317–337, 2016.
 - [35] E. Mullen, D. Zuniga, Z. Tatlock, and D. Grossman. Verified Peephole Optimizations for CompCert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 448–461, June 2016.
 - [36] A. Nötzli and F. Brown. LifeJacket: Verifying precise floating-point optimizations in LLVM. <http://arxiv.org/pdf/1603.09290v1.pdf>, 2016. Retrieved 2016-04-04.
 - [37] S. Padhi, R. Sharma, and T. Millstein. Data-driven Precondition Inference with Learned Features. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’16, pages 42–56, 2016.
 - [38] P. M. Phothisilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling Up Superoptimization. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 297–310, Apr. 2016.
 - [39] J. Regehr. Early Superoptimizer Results. <http://blog.regehr.org/archives/1146>. Retrieved 2016-11-14.
 - [40] J. Regehr. Signed Division and InstCombine. <http://lists.llvm.org/pipermail/llvm-dev/2016-June/100375.html>, 2016. Retrieved 2016-11-10.
 - [41] S. Sankaranarayanan, S. Chaudhuri, F. Ivančić, and A. Gupta. Dynamic Inference of Likely Data Preconditions over Predicates by Tree Learning. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA ’08, pages 295–306, 2008.
 - [42] E. R. Scherpelz, S. Lerner, and C. Chambers. Automatic Inference of Optimizer Flow Functions from Semantic Meanings. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 135–145, June 2007.
 - [43] E. Schkufza, R. Sharma, and A. Aiken. Stochastic Superoptimization. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, pages 305–316, 2013.
 - [44] M. N. Seghir and D. Kroening. Counterexample-Guided Precondition Inference. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP, pages 451–471, Mar. 2013.
 - [45] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori. A Data Driven Approach for Algebraic Loop Invariants. In *Proceedings of the 22nd European Conference on Programming Languages and Systems*, ESOP’13, pages 574–592, 2013.
 - [46] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial Sketching for Finite Programs. *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 404–415, Oct. 2006.
 - [47] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *PLDI ’10: Proceedings of the ACM SIGPLAN 2010 Conference on Programming Language Design and Implementation*, 2010.
 - [48] E. Torlak and R. Bodik. A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 530–541, June 2014.
 - [49] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and Understanding Bugs in C Compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, pages 283–294. ACM, 2011.

- [50] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM Intermediate Representation for Verified Program Transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 427–440, 2012.
- [51] J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formal Verification of SSA-Based Optimizations for LLVM. In *ACM SIGPLAN 2013 Conference on Programming Language Design and Implementation*, 2013.